quwsarohi@gmail.com Section: String Processing Page : 2

**// String Matching**

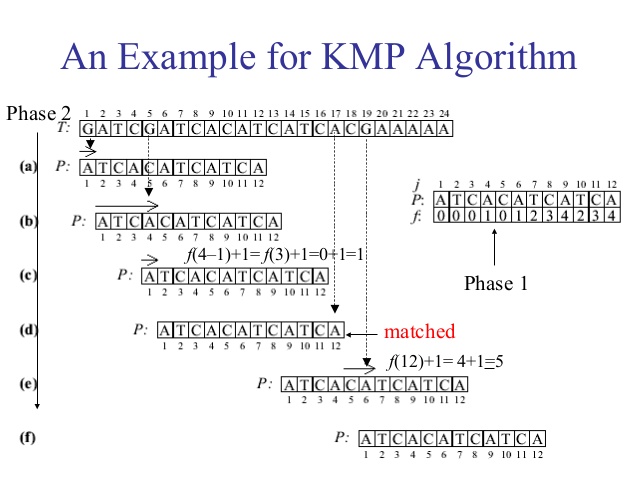
**// Knuth Morris Pratt**

**// Complexity : O(String + Token)**

char P[2000010], T[1000010]; //T is the string that we need to find

int P\_Size, T\_Size, Table[1000010]; //S is the string in which we have to find

void PrefixTable() { // Builds the prefix table

 int i = 0, j = -1; // Table contains the prefix table

Table[0] = -1;

while(i < T\_Size) { // Pre-process the pattern string T

while(j >= 0 && T[i] != T[j]) // If different, reset j using Table

j = Table[j]; // j = last point where i’th element = j’th element

i++, j++; // If same, advance both pointers

Table[i] = j;

} }

int KmpSearch() {

register int i = 0, j = 0, cnt = 0;

while(i < P\_Size) {

while(j >= 0 && P[i] != T[j]) // Search through string P

j = Table[j]; // If different, reset j using T

i++, j++; // if same, advance both pointers

if(j == T\_Size) { //the **match found** in i-j, if i-j = 0, then the whole string is matched

cnt++; // This happens when the string is equal in length of the token

//printf("%d'th Match found at %d\n", cnt, i-j); //the leftmost index

j = Table[j]; //j contains the first segment index that is matched in token

} }

return cnt; // Return the number of successful matches

**}**

**// Trie Basic**

**// Complexity : Build : O(S), Search : O(S)**

struct node {

//int visited; // Add if repeated substring needed

bool isEnd; // Indicates if this node contains a string that ends at this character

node \*next[11]; // How many child a root/parent node may contain

node() { // Initializer

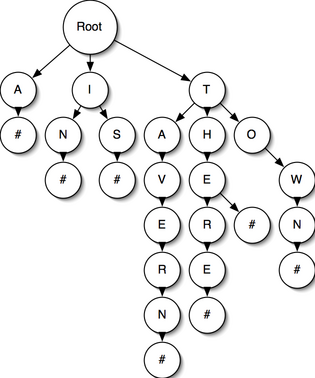
isEnd = false;

for(int i = 0; i < 10; i++)

next[i] = NULL;

}};

quwsarohi@gmail.com Section: String Processing Page : 2

bool create(char str[], int len, node \*current) { // Insert string in trie

for(int i = 0; i < len; i++) {

int pos = str[i] - '0';

if(current->next[pos] == NULL) // If this point don’t have child

current->next[pos] = new node(); // Initialize child

current = current - > next[pos];

current - >visited++; // Use this line if number of times visited in a node is

} //required

current->isEnd = true;

return false;

}

void del(node \*current) { // Deletes trie

for(int i = 0; i < 10; i++)

if(current->next[i] != NULL)

del(current→next[i]); Fig: Trie, # are isEnd = True

delete current;

}

bool found = 0;

void search(node \*current) {

for(int i = 0; i < 10; i++) {

if(current->next[i] != NULL)

check(current->next[i]);

}

if(found) return;

if(current->isEnd && !found) {

for(int i = 0; i < 10 && !found; i++)

if(current->next[i] != NULL) {

found = 1;

} } }

main() { …………

node\* root = new node(); // Creating root node

// Use this to build Prefix Trie

for(int i = 0; i < string\_len; i++)

create(Str+i, string\_len-i, 0, root); // Both LCS and LRS will need this

// To make trie with normal string

create(Str, strLen, root);

del(root);

……..}

**//Longest Repeated Substring**

**// Prefix Trie**

// ‘ATGATGAT’ : longest repeated substring : ‘ATG’

struct node {

int visited; // Indicates how many times this node is used

bool isEnd; // Indicates if this node contains a string that ends at this character

node \*next[4]; // How many child a root/parent node may contain

quwsarohi@gmail.com Section: String Processing Page : 3

node() { // Initialization

visited = 0;

isEnd = false;

for(int i = 0; i < 4; i++)

next[i] = NULL;

**}};**

string LongestRrepeatedSubstr(node \*current, string past) { // Longest Repeated Substring

int pos; // string past contains the past matched part

string longestRepeated; // y will contain the best repeated string longest and repeated

longestRepeated += past;

for(int i = 0; i < 4; i++) { **// Here every node contains four (4) child**

if(current->next[i] != NULL) { **// Change this line according to child**

if(current->next[i]->visited > 1) { // If this segment/char is visited more than once

string tmp;

tmp += map\_to\_str[i]; // Take this str as tmp

tmp += LRS(current->next[i], ""); // Find next LRS

if(tmp.size() > y.size()) // If the string found in this node is larger than previous found string

longestRepeated = tmp; // Take the largest

else if(tmp.size() == y.size()) // If both found in this search and the previous

if(tmp < longestRepeated) // If tmp is lexicographically smaller

longestRepeated = tmp; // take tmp as repeated substring

} } }

return longestRepeated; // LRS of ‘AG**AG**AG’ is 2, ‘AG**AG**’ and ‘**AG**AG’ both **AG** is common

}

main() { …….

string LRS = LongestRrepeatedSubstr(root, “”); // Gives the longest repeated substring

node \*current = root;

for(int i = 0; i < LRS.size(); i++) {

if(current - > next[ LRS[i]-’a’ ] != NULL) // lrs[i] – ‘a’ = index of that string

current = current - > next[ LRS[i]- ‘0’ ];

}

printf("%d\n", current - >visited); // Prints how many times the string is repeated

…….}

**//Longest Common Substring**

**// Prefix Trie**

// For two string Longest Common Substring is the longest substring that is the node is visited by two or more strings

// This code is for two LCS in two strings

struct node {

node \*next[5]; // How many child a root/parent node may contain

bitset<2>visited; // Indicates which string visited this node

node() { // Initialization

visited.reset();

for(int i = 0; i < 5; i++)

next[i] = NULL;

}};

quwsarohi@gmail.com Section: String Processing Page : 4

int max\_len = -1; ​​// Maximum length of substring is set to -1 by default

vector<string>lcs\_str; // This contains all the substring

void create(char str[], int len, int strNo, node \*current) { //Same as create in Trie

for(int i = 0; i < len; i++) { // Change **strNo** according to new strings

int pos = str[i] - ‘a’;

if(current->next[pos] == NULL)

current->next[pos] = new node();

current = current->next[pos];

current->visited[on] = 1; // Only this line is extra

}}

void longestCommonSubstrring(string past, node \*current, int totalStr) {

for(int i = 0; i < 4; i++) { **// Here every node contains four (4) child**

if(current->next[i] != NULL) { **// Change this line according to child**

if(current->next[i]->visited.count() == totalStr) { // If the node is visited from both strings

string tmp;

tmp += past; // Take past string + new found string

tmp += map\_str\_to\_int[i];

max\_len = max(max\_len, (int)tmp.size()); // Find the maximum length string

LCS(tmp, current - > next[i], totalStr); // Go for deeper match, this will add the deeper strings before this substr

if(tmp.size() == mx\_len) // If This substring is the longest

lsc\_str.push\_back(tmp); // push to lcs\_str

} } } }

main() {……..

for(int i = 0; i < len; i++) // Building Prefix Trie with string

build(S1+i, len-i, 0, root); // Change the **strNo** according to different string

for(int i = 0; i < len; i++)

build(S+i, len-i, 1, root); // **strNo** changed in other string

mx\_len = -1;

longestCommonSubstrring("", root, 2); // Here 2 is used as we are **finding LCS in two string**

for(int i = 0; i < lcs\_str.size(); i++)

if(lcs\_str[i].size() == mx\_len) //Only Printing the Longest Substring

printf("%s\n", lcs\_str[i].c\_str()); // Other substrings are also in this vector

del(root);

……….}

**Scanf Tricks :**

// %\* is used for skipping

// %[words that will be a valid input]

// %[^ words what will be invalid input, in this case, scanf will break]

scanf(" %\*[(] %[^+] %\*[+] %[^)] %s", a, b, n); // Input : (alpha+omega)^2 || a = alpha, b = omega = n = 2

// %\*[(] skipping (

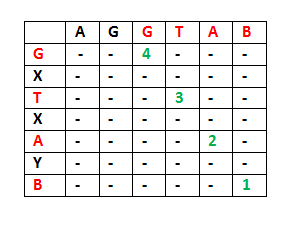
// %[^+] take input until +

// %\*[+] skipping +

// %\*[^)] skipping ^ and )

**Empty Line Input:** If the input contains empty lines that also should be processed, use fgets()

quwsarohi@gmail.com Section: String Processing Page : 5

**// Longest Common Subsequence (Not Substring!)**

**// Complexity : O(len\_a\*len\_b) (Dynamic Programming)**

// Bottom Up DP

int LCS(char a, char b, int len\_a, int len\_b) {

int dp[210][210];

for(register int i = 0; i <= len\_a; i++)

for(register int j = 0; j <= len\_b; j++) {

if(i == 0 || j == 0) //base case

dp[i][j] = 0;

else if(a[i-1] == b[j-1]) //if a match found Fig: Longest Common Substring DP table

dp[i][j] = dp[i-1][j-1] + 1;

else

dp[i][j] = max(dp[i-1][j], dp[i][j-1]); // dp[i][j] = max(ignoring a[i-1] (taking b[j]), (taking a[i]) ignoring b[j-1])

}

return dp[len\_a][len\_b];

}

**// Longest Palindrom**

**// Dynamic Programming**

char S[1010];

int dp[1010][1010], len;

int palindrom(int l, int r) { // function call: palindrom(0, length\_of\_string)

if(dp[l][r] != -1) //memorization

return dp[l][r];

else if(l == r) //if the middle point reached (odd length of a string)

return dp[l][r] = 1;

else if(l+1 == r) { //if the two points are middle (even length of a string)

if(S[l] == S[r])

return dp[l][r] = 2; //if matches, we can take them both

else

return dp[l][r] = 1; //else we can take only one of them

}

else {

if(S[l] == S[r]) //if the first and the last character is matched, then we can take them both and go deeper

dp[l][r] = 2 + palindrom(l+1, r-1);

else //else we will search for the best choice

dp[l][r] = max(palindrom(l+1, r), palindrom(l, r-1));

}

return dp[l][r];

}